XPi: a typed process calculus for XML messaging*

Lucia Acciai[1] and Michele Boreale[2]

[1] Laboratoire d'Informatique Fondamentale de Marseille, Université de Provence.
[2] Dipartimento di Sistemi e Informatica, Università di Firenze.
lucia.acciai@lif.univ-mrs.fr, boreale@dsi.unifi.it

Abstract. We present XPi, a core calculus for XML messaging. XPi features asynchronous communications, pattern matching, name and code mobility, integration of static and dynamic typing. Flexibility and expressiveness of this calculus are illustrated by a few examples, some concerning description and discovery of web services. In XPi, a type system disciplines XML message handling at the level of channels, patterns, and processes. A run-time safety theorem ensures that in well-typed systems no service will ever receive documents it cannot understand, and that the offered services, even if redefined, will be consistent with the declared channel capacities.

1 Introduction

The design of globally distributed systems, like Web Services (WS, [23]) or business-to-business applications [5], is more and more centered around passing of messages in the form of XML documents. Major reasons for the emergence of message-passing are its conceptual simplicity, its minimal infrastructural requirements, and its neutrality with respect to back-ends and platforms of services [6]. These features greatly ease interoperability and integration.

It is generally recognized that some of the proposed languages and standards for WS draw their inspiration from the π-calculus [19]. The latter conveys the message-passing paradigm in a distilled form. In practice, at one extreme we find languages like WSDL [12], useful to describe service interfaces, but saying very little about behaviour. At the other extreme, we find proposed standards like BPEL4WS [2], oriented to detailed descriptions of services, but hardly amenable to formal analysis. In other words, we are experiencing a significant gap between theory (formal models and analysis techniques) and practice (programming) in the field of distributed applications.

As a first step toward filling this gap, we aim at giving a concise semantic account of XML messaging and of the related typing issues. To this purpose, we present *XPi*, a process language based on the asynchronous π-calculus. Prominent features of XPi are: patterns generalizing ordinary inputs, ML-like pattern matching, and integration of static and dynamic typing. Our objective is to study issues raised by these features in connection with name and code mobility. A more precise account of our work and contributions follows.

For the sake of simplicity, syntax and reduction semantics of untyped XPi are first introduced (Section 2). In XPi, resource addresses on the net are represented as *names*, which can be generally understood as channels at which services are listening. *Messages* passed around are XML documents, represented as tagged/nested lists, in the vein of XDuce [15, 16]. Services and their clients are *processes*, that may send messages to channels, or query channels to retrieve messages obeying given patterns. Messages may contain names, which are passed around with only the *output capability* [20]. Practically, this means that a client receiving a service address

* This work has been partially supported by EU within the IST FET - Global Computing initiative, projects MIKADO and PROFUNDIS.

cannot use this address to re-define the service. This assumption is perfectly sensible, simplifies typing issues, and does not affect expressive power (see e.g. [7, 17]). Messages may also contain mobile code in the form of *abstractions*, roughly, functions that take some argument and yield a process as a result. More precisely, abstractions can consume messages through pattern matching, thus supplying actual parameters to the contained code and starting its execution. This mechanism allows for considerable expressiveness. For example, we show that it permits a clean encoding of encryption primitives, hence of the spi-calculus [1], into XPi.

Types (Section 3) discipline processing of messages at the level of channels, patterns, and processes. At the time of its creation, each channel is given a *capacity*, i.e. a type specifying the format of messages that can travel on that channel. *Subtyping* arises from the presence of star types (arbitrary length lists) and union types, and by lifting at the level of messages a subtyping relation existing on basic values. The presence of a top type $\mathbf{T}$ enhances flexibility, allowing for such types as "all documents with an external tag f, containing a tag g and something else", written $\mathsf{T} = f[g[\mathbf{T}], \mathbf{T}]$. Subtyping is contravariant on channels: this is natural if one thinks of services, roughly, as functions receiving their arguments through channels. Contravariance calls for a bottom type $\mathbf{\bot}$, which allows one to express such sets of values as "all channels that can transport documents of some type $\mathsf{S} < \mathsf{T}$", written $ch(f[g[\mathbf{\bot}], \mathbf{\bot}])$. Abstractions that can safely consume messages of type T are given type $(\mathsf{T})\mathsf{Abs}$. Interplay between pattern matching, types, and capacities raises a few interesting issues concerning *type safety* (Section 4). Stated in terms of services accessible at given channels, our run-time safety theorem ensures that in well-typed systems, first, no service will ever receive documents it cannot understand, and second, that the offered service, even when re-defined, will comply with the statically declared capacities. The first property simply means that no process will ever output messages violating channel capacities. The second property means that no service will hang due to a input pattern that is not consistent with the channel's capacity (a form of "pattern consistency"). Type checking is entirely static, in the sense that no run-time type check is required.

Our type system is partially inspired by XSD [13], but is less rich than, say, the language of [9]. In particular, we have preferred to omit recursive types. While certainly useful in a full blown language, recursion would raise technicalities that hinder issues concerning name and code mobility. Also, our pattern language is quite basic, partly for the similar reasons of simplicity, partly because more sophisticated patterns can be easily simulated.

The calculus described so far enforces a strictly static typing discipline. We also consider an extension of this calculus with *dynamic abstractions* (Section 5), which are useful when little or nothing is known about the actual types of incoming messages. Run-time type checks ensure that substitutions arising from pattern matching respect the types statically assigned to variables. Run time safety carries over. We shall argue that dynamic abstractions, combined with code mobility and subtyping, can provide linguistic support to such tasks as publishing and querying services.

There have been a number of proposals for integrating XML manipulation primitives into statically typed languages. We conclude (Section 6) with some discussion on recent related work in this field, and with a few directions for future extensions.

2 Untyped XPi

Syntax We assume a countable set of *variables* $\mathcal{V}$, ranged over by $x, y, z, \ldots$, a set of *tags* $\mathcal{F}$, ranged over $f,g,\ldots$, and a set of *basic values* $\mathcal{BV}$ $v, w, \ldots$. We leave $\mathcal{BV}$ unspecified (it might contain such values as integers, strings, or Java objects), but assume that $\mathcal{BV}$ contains a countable set of *names* $\mathcal{N}$, ranged over by $a, b, c, \ldots$. $\mathcal{N}$ is partitioned into a family of countable sets called *sorts* $\mathcal{S}, \mathcal{S}', \ldots$. We let u range over $\mathcal{N} \cup \mathcal{V}$ and $\tilde{x}, \ldots$ denote a tuples of variables.

Message	$M ::=$	v	*Value*
		$\mid x$	*Var*
		$\mid f(M)$	*Tag*
		$\mid LM$	*List*
		$\mid A$	*Abstraction*
List of messages	$LM ::=$	$[]$	*Empty list*
		$\mid x$	*Var*
		$\mid M \cdot LM$	*Concatenation*
Abstraction	$A ::=$	$(Q_{\tilde{x}})P$	*Pattern and Continuation*
		$\mid x$	*Var*
Pattern	$Q ::=$	v	*Value*
		$\mid x$	*Var*
		$\mid f(Q)$	*Tag*
		$\mid LQ$	*List*
List of patterns	$LQ ::=$	$[]$	*Empty list*
		$\mid x$	*Var*
		$\mid Q \cdot LQ$	*Concatenation*
Process	$P ::=$	$\overline{u}\langle M \rangle$	*Output*
		$\mid \sum_{i \in I} a_i.A_i$	*Guarded Summation*
		$\mid P \text{ else } R$	*Else*
		$\mid P_1 \mid P_2$	*Parallel*
		$\mid !P$	*Replication*
		$\mid (\nu a)P$	*Restriction*

Table 1. Syntax of XPi messages, patterns and processes.

Definition 1 (messages, patterns and processes). *The set $\mathcal{M}$ of XPi* messages $M, N, \ldots$, *the set $\mathcal{Q}$ of XPi* patterns $Q, Q', \ldots$ *and the set $\mathcal{P}$ of XPi* processes $P, R, \ldots$ *are defined by the syntax in Table 1. In $Q_{\tilde{x}}$, we impose the following linearity condition: $\tilde{x}$ is a tuple of distinct names and each $x_i \in \tilde{x}$ occurs at most once in Q.*

In the style of XDuce [15, 16] and ℂDuce [3] XML documents are represented in XPi as tagged ordered list that can be arbitrarily nested; these are the messages being exchanged among processes. A message can be either a basic value, a variable, a tagged message, a list of messages, or an abstraction. The latter take the form $(Q_{\tilde{x}})P$, where variables $\tilde{x}$ represent formal parameters, to be replaced by actual parameters at run-time. A pattern is simply an abstraction-free message. For the sake of simplicity, we have ignored tag-variables that could be easily accommodated. Also, note that patterns do not allow for direct decomposition of documents into sublists (akin to the pattern `p, p'` in XDuce). The latter can be easily encoded though, as we show later in this section.

Process syntax is a variation on the π-calculus. In particular, asynchronous (non blocking) output on a channel u is written $\overline{u}\langle M \rangle$, and u is said to occur in *output subject position*.

Nondeterministic guarded summation $\sum_{i\in I} a_i.A_i$ waits for any message matching A_i's pattern at channel a_i, for some $i \in I$, consumes this message and continues as prescribed by A_i; names a_i are said to occur in *input subject position*. Note that the syntax forbids variables in input subject position, hence a received name cannot be used as an input channel; in other words, names are passed around with the output capability only. Parallel composition $P_1|P_2$ represents concurrent execution of P_1 and P_2. Process P else R behaves like P, if P can do some internal reduction, otherwise reduces to R. This operator will be useful for coding up, e.g., if-then-else, without the burden of dealing with explicit negation on pattern. Replication $!P$ represents the parallel composition of arbitrarily many copies of P. Restriction $(\nu a)P$ creates a fresh name a, whose initial scope is P. Usual binding conventions and notations (alpha equivalence $=_\alpha$, free and bound names $\mathsf{fn}(\cdot)$ and $\mathsf{bn}(\cdot)$, free and bound variables $\mathsf{fv}(\cdot)$ and $\mathsf{bv}(\cdot)$) apply. We let $\mathcal{M}_{cl}$ be the set of closed messages and $\mathcal{P}_{cl}$ be the set of closed processes.

Notations The following abbreviations for messages and patterns are used: $[M_1, M_2, \ldots, M_{k-1}, M_k]$ stands for $M_1 \cdot (M_2 \cdot (\ldots (M_{k-1} \cdot (M_k \cdot [])) \ldots))$, while $f[M_1, M_2, \ldots, M_{k-1}, M_k]$ stands for $f([M_1, M_2, \ldots, M_{k-1}, M_k])$. The following abbreviations for processes are used: $\mathbf{0}$, $a_1.A_1$ and $a_1.A_1 + a_2.A_2 + \cdots + a_n.A_n$ stand for $\sum_{\{i\in I\}} a_i.A_i$ when $|I| = 0$, $|I| = 1$, and $|I| = n$, respectively; $(\nu\, a_1, \ldots, a_n)P = (\nu \widetilde{a})P$ stands for $(\nu a_1) \ldots (\nu a_n)P$. We sometimes save on subscripts by marking binding occurrences of variables in abstractions by a '?' symbol, or by replacing a binding occurrence of a variable by a don't care symbol, '_', if that variable does not occur in the continuation process. E.g. $([f[?x], g[\_]])P$ stands for $([f[x], g[y]]_{\{x,y\}})P$ where $y \notin \mathrm{fv}(P)$.

Our list representation of XML ignores algebraic properties of concatenation (such as associativity, see [16]). We simply take for granted some translation from actual XML documents to our syntax. The following example illustrates informally what this translation might look like.

Example 1. An XML document encoding an address book (on the left) and its representation in XPi (on the right)[1]:

```
<addrbook>
    <person>
        <name>John Smith</name>
        <tel>12345</tel>
        <emailaddrs>
           <email>john@smith</email>
           <email>smith@john</email>
        </emailaddrs>
    </person>
    <person>
        <name>Eric Brown</name>
        <tel>678910</tel>
        <emailaddrs></emailaddrs>
    </person>
</addrbook>.
```

```
addrbook[ person[ name("John Smith"),
                  tel(12345),
                  emailaddrs[email("john@smith"),
                             email("smith@john")]
                ],
          person[ name("Eric Brown"),
                  tel(678910),
                  emailaddrs[]
                ]
        ]
```

Note that a sequence of tagged documents such as `<tag1>M</tag1><tag2>N</tag2>`$\cdots$ is rendered as an ordered list `[tag1(M), tag2(N),`$\cdots$`]`. A pattern that extracts name and telephone number of the first person of the address book above is: $\mathtt{Q}_{\mathtt{xy}}$ = `addrbook[ person[name(?x),tel(?y),_],_]`.

[1] We shall prefer the typewriter font whenever useful to improve on readability.

$$
\begin{aligned}
P =_\alpha R &\Rightarrow P \equiv R \\
P|R &\equiv R|P \\
(P|R_1)|R_2 &\equiv P|(R_1|R_2) \\
P|\mathbf{0} &\equiv P \\
!P &\equiv P|!P \\
(\nu a)(P|R) &\equiv P|(\nu a)R \quad \text{if} \quad a \notin \mathrm{fn}(P) \\
(\nu a)\mathbf{0} &\equiv \mathbf{0} \\
(\nu a)(\nu b)P &\equiv (\nu b)(\nu a)P
\end{aligned}
$$

Table 2. Structural congruence.

$$
(\textsc{com}) \quad \frac{j \in I \quad a_j = a, \quad A_j = (Q_{\tilde{x}})P, \quad \mathsf{match}(M, Q, \sigma)}{\overline{a}\langle M\rangle \mid \sum_{i \in I} a_i.A_i \to P\sigma}
$$

$$
(\textsc{struct}) \quad \frac{P \equiv P', \quad P' \to Q', \quad Q' \equiv Q}{P \to Q} \qquad (\textsc{ctx}) \quad \frac{P \to P'}{(\nu \tilde{a})(P|R) \to (\nu \tilde{a})(P'|R)}
$$

$$
(\textsc{else}_1) \quad \frac{P \to P'}{P \text{ else } Q \to P'} \qquad (\textsc{else}_2) \quad \frac{P \not\to}{P \text{ else } Q \to Q}
$$

Table 3. Reduction semantics.

Reduction semantics A *reduction relation* describes system evolution via internal communications. Following [18], XPi reduction semantics is based on *structural congruence* $\equiv$, defined as the least congruence on processes satisfying the laws in Table 2. The latter permit certain rearrangements of parallel composition, replication, and restriction. The relation $\equiv$ extends to abstractions, hence to messages, in the expected manner. The reduction semantics also relies on a standard matching predicate, that matches a (linear) pattern against a closed message and yields a substitution.

Definition 2 (substitutions and matching). Substitutions $\sigma, \sigma', \ldots$ *are finite partial maps from the set* $\mathcal{V}$ *of variables to the set* $\mathcal{M}_{cl}$ *of closed messages. We denote by* ε *the empty substitution. For any term* t*,* $t\sigma$ *denotes the result of applying* σ *onto* t *(with alpha-renaming of bound names and variables if needed). Let* M *be a closed message and* Q *be a linear pattern:* $\mathsf{match}(M, Q, \sigma)$ *holds true if and only if* $\mathsf{dom}(\sigma) = \mathrm{fv}(Q)$ *and* $Q\sigma = M$*; in this case, we also say that* M matches Q*.*

Definition 3 (reduction). *The reduction relation,* $\to \subseteq \mathcal{P}_{cl} \times \mathcal{P}_{cl}$*, is the least binary relation on closed processes satisfying the rules in Table 3.*

Derived constructs and examples XPi allows for straightforward definition of a few powerful constructs, that will be used in later examples. In the following, we shall freely use recursive definitions of processes, that can be coded up using replication [18].

- *Application.* A functional-like application for abstractions, $A \bullet M$, can be defined as $(\nu c)(\overline{c}\langle M\rangle | c.A)$, for any $c \notin \mathrm{fn}(M, A)$.
- *Case.* A pattern matching construct relying on a *first match* policy, written

$$
\text{Case } M \text{ of} \quad (Q_1)_{\tilde{x}_1} \Rightarrow P_1, \ (Q_2)_{\tilde{x}_2} \Rightarrow P_2, \cdots, (Q_k)_{\tilde{x}_k} \Rightarrow P_k
$$

  evolves into P_1 if M matches Q_1 (with substitutions involved), otherwise evolves into P_2 if M matches Q_2, and so on; if there is no match, the process is stuck. This construct can be defined in XPi as follows (assuming precedence of $\bullet$ on else and right-associativity for else):

$$(Q_1)_{\tilde{x}_1} P_1 \bullet M \quad \text{else} \quad (Q_2)_{\tilde{x}_2} P_2 \bullet M \quad \text{else} \quad \cdots \quad \text{else} \quad (Q_k)_{\tilde{x}_k} P_k \bullet M\,.$$

– *Decomposition.* A process that attempts to *decompose* a message M into two sublists that satisfy the patterns $Q_{\tilde{x}}$ and $Q'_{\tilde{y}}$ and proceeds like P (with substitutions for $\tilde{x}$ and $\tilde{y}$ involved), if possible, otherwise is stuck, written: M as $Q_{\tilde{x}}, Q'_{\tilde{y}} \Rightarrow P$, can be defined as the recursive process $R([\,[\,], M])$, where:

$$\begin{aligned} R([l,x]) = \text{Case } x \text{ of } ?y \cdot ?w \Rightarrow (\,\text{Case } l@y \text{ of } Q_{\tilde{x}} \Rightarrow (\,\text{Case } w \text{ of} \quad & Q'_{\tilde{y}} \Rightarrow P, \\ & \_ \Rightarrow R([l@y,\, w])\,), \\ \_ \Rightarrow\ & R([l@y, w])\,). \end{aligned}$$

Here we have used a list-append function @, which can be easily defined via a call to a suitable recursive process. Most common list manipulation constructs can be easily coded up in this style. We shall not pursue this direction any further.

Example 2 (a web service). Consider a web service WS that offers two different services: an audio streaming service, offered at channel *stream*, and a download service, offered at channel *download*. Clients that request the first kind of service must specify a streaming channel and its bandwidth (`"high"` or `"low"`), so that WS can stream one of two mp3 files ($\mathtt{v}_{low}$ or $\mathtt{v}_{high}$), as appropriate. Clients that request download must specify a channel at which the player will be received. A client can run the downloaded player locally, supplying it appropriate parameters (a local streaming channel and its bandwidth). We represent streaming on a channel simply as an output action along that channel:

$$\begin{aligned} WS \stackrel{\triangle}{=} !(\quad & \mathit{stream}.(\mathtt{req\_stream[bandwidth("low"),channel(?}x\mathtt{)]})\overline{x}\langle \mathtt{v}_{low} \rangle \\ & + \mathit{stream}.(\mathtt{req\_stream[bandwidth("high"),channel(?}y\mathtt{)]})\overline{y}\langle \mathtt{v}_{high} \rangle \\ & + \mathit{download}.(\mathtt{req\_down(?}z\mathtt{)})\overline{z}\langle \mathit{Player} \rangle \quad). \end{aligned}$$

Player is an abstraction:

$$\begin{aligned} \mathit{Player} \stackrel{\triangle}{=} (\mathtt{req\_stream[bandwidth(?}y\mathtt{),channel(?}z\mathtt{)]})(\ \text{Case } y \text{ of } & \mathtt{"low"} \Rightarrow \overline{z}\langle \mathtt{v}_{low} \rangle \\ & \mathtt{"high"} \Rightarrow \overline{z}\langle \mathtt{v}_{high} \rangle \quad). \end{aligned}$$

Note that the first two summands of WS are equivalent to *stream.Player*. However, the extended form written above makes it possible a static optmization of channels (see Example 5).
A client that asks for low bandwidth streaming, listens at s and then proceeds like C is:

$$C_1 \stackrel{\triangle}{=} (\nu s)(\overline{\mathit{stream}}\langle \mathtt{req\_stream[bandwidth("low"),channel(}s\mathtt{)]} \rangle \,|\, s.(?v)C).$$

Another client that asks for download, then runs the player locally, listening at a local high bandwidth channel s is C_2 defined as:

$$\begin{aligned} (\nu d, s)(\quad & \overline{\mathit{download}}\langle \mathtt{req\_down(}d\mathtt{)} \rangle \,|\, d.(?x_p)(x_p \bullet \\ & \mathtt{req\_stream[bandwidth("high"),channel(}s\mathtt{)]} \,|\, s.(?v)C) \quad). \end{aligned}$$

Encryption and decryption Cryptographic primitives are sometimes used in distributed applications to guarantee secrecy and authentication of transmitted data. As a testbed for expressiveness, we show how to encode shared-key encryption and decryption primitives à la spi-calculus [1] into XPi. We shall see an example of application of these encodings in Section 5. We first introduce XPi$^{\mathrm{cr}}$, a cryptographic extension of XPi that subsumes shared-key spi-calculus, and then show how to encode XPi$^{\mathrm{cr}}$ into XPi. Message syntax is extended with the following clause, that represents encryption of M using N as a key:

$$M \quad ::= \quad \cdots \mid \{M\}_N \qquad \textit{(encryption)}$$

where N does contain neither abstractions nor encryptions. Process syntax is extended with a case operator, that attempts decryption of M using N as a key and if successful binds the result to a variable x :

$$P \quad ::= \quad \cdots \mid \text{case } M \text{ of } \{x\}_N \text{ in } P \qquad \textit{(decryption)}$$

where N does contain neither abstractions nor encryptions, M is a variable or a message of the form $\{M'\}_{N'}$ and x binds in P. Patterns remain unchanged, in particular they may not contain encryptions or abstractions. The additional reduction rule is:

$$(\textsc{Dec}) \quad \text{case } \{M\}_N \text{ of } \{x\}_N \text{ in } P \to P[M/x].$$

Next, two translation functions, one for messages ($[\![\cdot]\!]$) and one for processes ($(\!|\cdot|\!)$), are defined from XPi$^{\text{cr}}$ to XPi. The translations of messages follow a familiar continuation-passing style. The relevant clauses of the definition, by structural induction, are as follows (on the others the functions just go through the structure of terms):

$$[\![u]\!] = u$$
$$[\![\{M\}_N]\!] = ([N, ?x])\,\overline{x}\langle [\![M]\!] \rangle$$

$$(\!|\overline{u}\langle M\rangle|\!) = \overline{u}\langle [\![M]\!] \rangle$$
$$(\!|\text{case } M \text{ of } \{x\}_N \text{ in } P|\!) = (\nu\, r)\,([\![M]\!] \bullet [N, r] \mid r.(?x)\,(\!|P|\!)).$$

Following [22], let us define the barb predicate $P \Downarrow a$ as follows: there is P' s.t. $P \to^* P'$ and P' has either an input summand $a.A$ or an output $\overline{a}\langle M\rangle$ which are not in the scope of a (νa), an else or guarded summation. The encoding defined above is correct, in the sense that it preserves reductions and barbs in both directions, as stated by the proposition below. Note that, by compositionality, this implies the encoding is fully abstract w.r.t. barbed equivalence (see e.g. [7]).

Proposition 1. *Let P be a closed process in XPi$^{\text{cr}}$.*

1. *if $P \to P'$ then $(\!|P|\!) \to^* (\!|P'|\!)$;*
2. *if $(\!|P|\!) \to P'$ then $\exists P'' \in XPi^{\text{cr}}$ s.t. $P' \to^* (\!|P''|\!)$;*
3. *$P \Downarrow a$ if and only if $(\!|P|\!) \Downarrow a$.*

3 A Type System

In this section, we define a type system for XPi that disciplines messaging at the level of channels, patterns and processes. The system guarantees that well-typed processes respect channels capacities at runtime. In other words, services are guaranteed to receive only requests they can understand, and conversely, services offered at a given channel will be consistent with the type declared for that channel. XPi's type system draws its inspiration from, but is less rich than, XML-Schema [13]. Our system permits to specify types for basic values (such as string or int) and provides tuple types (fixed-length lists) and star types (arbitrary-length lists); moreover, it provides abstraction types for code mobility. For the sake of simplicity, we have omitted attributes and recursive types.

Types	$\mathsf{T} ::=$	bt	*Basic type* ($\mathsf{bt} \in \mathcal{BT}$)
		$\mid \mathbf{T}$	*Top*
		$\mid \mathbf{\perp}$	*Bottom*
		$\mid f(\mathsf{T})$	*Tag* ($f \in \mathcal{F}$)
		$\mid \mathsf{LT}$	*List*
		$\mid \mathsf{T}+\mathsf{T}$	*Union*
		$\mid (\mathsf{T})\mathsf{Abs}$	*Abstraction*
List types	$\mathsf{LT} ::=$	$[]$	*Empty*
		$\mid *\mathsf{T}$	*Star*
		$\mid \mathsf{T} \cdot \mathsf{LT}$	*Concatenation*

Table 4. Syntax of types.

Message types and subtyping We assume an unspecified set of *basic types* $\mathcal{BT}$ bt, bt',... that might include int, string, boolean, or even Java classes. We assume that $\mathcal{BT}$ contains a countable set of *sort names* in one-to-one correspondence with the sorts $\mathcal{S}, \mathcal{S}', ...$ of $\mathcal{N}$; by slight abuse of notation, we denote sort names by the corresponding sorts.

Definition 4 (types). *The set $\mathcal{T}$ of* types*, ranged over by T, S, ..., is defined by the syntax in Table 4.*

Note the presence of the union type $\mathsf{T}+\mathsf{T}'$, that is the type of all messages of type T or T', and of the star type $*\mathsf{T}$, that is the type of all lists of elements of type T. $(\mathsf{T})\mathsf{Abs}$ is the type of all abtractions that can consume messages of type T. Finally, note the presence of $\mathbf{T}$ and $\mathbf{\perp}$ types. $\mathbf{T}$ is simply the type of all messages. On the contrary, no message has type $\mathbf{\perp}$, but this type is extremely useful for the purpose of defining channel types, as we shall see below.

Notation The following abbreviations for types are used: $[T_1, T_2, \ldots, T_{k-1}, T_k]$ stands for $T_1 \cdot (T_2 \cdot (\ldots (T_{k-1} \cdot (T_k \cdot [])) \ldots))$, while $f[T_1, T_2, \ldots, T_{k-1}, T_k]$ stands for $f([T_1, T_2, \ldots, T_{k-1}, T_k])$.

Example 3. A type for address books, on the left (see message M in Example 1), and a type for all SOAP messages, consisting of an *optional* header and a body, enclosed in an envelope, on the right:

```
addrbook[ *person[ name(string),                    envelope[ [] + header(T),
                   tel(int),                                  body(T)
                   emailaddrs(*email(string))]]          ].
```

Next, we associate types with channels, or more precisely with sorts. This is done by introducing a "capacity" function.

Definition 5. *A* capacity *function is a surjective map from the set of sorts to the set of types.*

In the sequel, we fix a generic capacity function. We shall denote by $ch(\mathsf{T})$ a generic sort that is mapped to T. Note that, by surjectivity of the capacity function, for each type T there is a sort $ch(\mathsf{T})$. In particular, $ch(\mathbf{T})$ is the sort of channels that can transport anything. In practice, determining capacity T of a given channel a, i.e. that a belongs to $ch(\mathsf{T})$, might be implemented with a variety of mechanisms, such as attaching to a an explicit reference to T's definition. We abstract away from these details.

List and star types and the presence of $\mathbf{T}$ and $\mathbf{\perp}$ naturally induce a subtyping relation. For example, a service capable of processing messages of type $\mathsf{T} = f(* \mathsf{int})$ must be capable of

$$\text{(Sub-Sort)}\quad \frac{\mathsf{T} < \mathsf{T}'}{ch(\mathsf{T}') < ch(\mathsf{T})}$$

$$\text{(Sub-Top)}\quad \frac{}{\mathsf{T} < \mathbf{T}} \qquad \text{(Sub-Bottom)}\quad \frac{}{\mathbf{\bot} < \mathsf{T}}$$

$$\text{(Sub-Basic)}\quad \frac{\mathsf{bt1} \prec \mathsf{bt2}}{\mathsf{bt1} < \mathsf{bt2}} \qquad \text{(Sub-Tag)}\quad \frac{\mathsf{T}' < \mathsf{T}}{f(\mathsf{T}') < f(\mathsf{T})}$$

$$\text{(Sub-Star}_1\text{)}\quad \frac{}{[\,] < *\mathsf{T}} \qquad \text{(Sub-Star}_2\text{)}\quad \frac{\mathsf{T}' < \mathsf{T}, \quad \mathsf{LT} < *\mathsf{T}}{\mathsf{T}' \cdot \mathsf{LT} < *\mathsf{T}}$$

$$\text{(Sub-Star}_3\text{)}\quad \frac{\mathsf{T}' < \mathsf{T}}{*\mathsf{T}' < *\mathsf{T}} \qquad \text{(Sub-List)}\quad \frac{\mathsf{T}_1 < \mathsf{T}'_1, \quad \mathsf{LT} < \mathsf{LT}'}{\mathsf{T}_1 \cdot \mathsf{LT} \ < \mathsf{T}'_1 \cdot \mathsf{LT}'}$$

$$\text{(Sub-Union}_1\text{)}\quad \frac{\mathsf{T} < \mathsf{T}' \text{ or } \mathsf{T} < \mathsf{T}''}{\mathsf{T} < \mathsf{T}' + \mathsf{T}''} \qquad \text{(Sub-Union}_2\text{)}\quad \frac{\mathsf{T}' < \mathsf{T}, \quad \mathsf{T}'' < \mathsf{T}}{\mathsf{T}' + \mathsf{T}'' < \mathsf{T}}$$

Table 5. Rules for subtyping.

processing messages of type $\mathsf{T}' = f[\mathsf{int}, \mathsf{int}]$, i.e. T' is a subtype of T. Subtyping also serves to lift a generic subtyping preorder on basic types, $\prec$, to all types.

Definition 6 (subtyping). *The* subtyping relation $< \subseteq \mathcal{T} \times \mathcal{T}$ *is the least reflexive and transitive relation closed under the rules of Table 5.*

Note that we disallow subtyping on abstractions. The reason for this limitation will be discussed shortly after presenting the type checking system (see Remark 1). Also note that subtyping is contravariant on sorts capacities (rule (Sub-Sort)): this is natural if one thinks of a name of capacity T as, roughly, a function that can take arguments of type T. As a consequence of contravariance, for any T, we have $ch(\mathsf{T}) < ch(\mathbf{\bot})$, that is, $ch(\mathbf{\bot})$ is the type of all channels.

Type checking A *basic typing* relation $v : \mathsf{bt}$ on basic values and basic types is presupposed, which is required to respect subtyping, i.e. whenever $\mathsf{bt} \prec \mathsf{bt}'$ and $v : \mathsf{bt}$ then $v : \mathsf{bt}'$. We further require that for each bt there is at least one $v : \mathsf{bt}$, and that for each v the set of bt's s.t. $v : \mathsf{bt}$ has a minimal element. On names and sort names the basic typing relation is the following: $a : \mathcal{S}$ iff $a \in \mathcal{S}'$ for some $\mathcal{S}' < \mathcal{S}$.

Contexts $\Gamma, \Gamma', \ldots$ are finite partial maps from variables $\mathcal{V}$ to types $\mathcal{T}$, sometimes denoted as sets of variable bindings $\{x_i : \mathsf{T}_i\}_{i \in I}$ (x_i's distinct). We denote the empty context by $\emptyset$. Assume $\widetilde{x}$ a set of variables; we denote by $\Gamma_{-\widetilde{x}}$ the context obtained from Γ by removing the bindings for the variables in $\tilde{x}$, and by $\Gamma_{|\widetilde{x}}$ the context obtained by restricting Γ to the bindings for the variables in $\tilde{x}$. The subtyping relation is extended to contexts by letting $\Gamma_1 < \Gamma_2$ iff $\mathsf{dom}(\Gamma_1) = \mathsf{dom}(\Gamma_2)$ and $\forall x \in \mathsf{dom}(\Gamma_1)$ it holds that $\Gamma_1(x) < \Gamma_2(x)$. Union of contexts Γ_1 and Γ_2 having disjoint domains is written as $\Gamma_1 \cup \Gamma_2$ or as Γ_1, Γ_2 if no ambiguity arises. Sum of contexts Γ_1 and Γ_2 is written as $\Gamma_1 + \Gamma_2$ and is defined as $(\Gamma_1 + \Gamma_2)(x) = \Gamma_1(x) + \Gamma_2(x)$ if $x \in \mathsf{dom}(\Gamma_1) \cap \mathsf{dom}(\Gamma_2)$, otherwise $(\Gamma_1 + \Gamma_2)(x) = \Gamma_i(x)$ if $x \in \mathsf{dom}(\Gamma_i)$ for $i = 1, 2$.

Type checking relies on a type-pattern matching predicate, $\mathsf{tpm}(\mathsf{T}, Q, \Gamma)$, whose role is twofold: (1) it extracts from T the types expected for variables in Q after matching against messages of type T, yielding the context Γ, (2) it checks that Q is consistent with type T, i.e. that the type of Q is of a subtype of T under Γ.

Definition 7 (type-pattern match). *The predicate* $\mathsf{tpm}(T, Q, \Gamma)$ *is defined by the rules in Table 6.*

As expected, type checking works on an *annotated syntax*, where each $Q_{\widetilde{x}}$ is decorated by a context Γ for its binding variables $\widetilde{x}$, written $Q_{\widetilde{x}} : \Gamma$, with $\tilde{x} = \mathsf{dom}(\Gamma)$, or simply $Q : \Gamma$, where it is understood that the binding variables of Q are $\mathsf{dom}(\Gamma)$. For notational simplicity, we shall use

$$(\text{TPM-TOP}) \quad \frac{Q \neq x}{\text{tpm}(\mathbf{T}, Q, \Gamma)},\ \forall x \in \text{fv}(Q) :\ \Gamma(x) = \mathbf{T}$$

$$(\text{TPM-EMPTY}) \ \frac{}{\text{tpm}([],[],\emptyset)} \quad (\text{TPM-VAR}) \ \frac{}{\text{tpm}(\mathsf{T}, x, \{x:\mathsf{T}\})}$$

$$(\text{TPM-VALUE}) \ \frac{v : \mathsf{bt}}{\text{tpm}(\mathsf{bt}, v, \emptyset)} \quad (\text{TPM-TAG}) \ \frac{\text{tpm}(\mathsf{T}, Q, \Gamma)}{\text{tpm}(f(\mathsf{T}), f(Q), \Gamma)}$$

$$(\text{TPM-STAR}_1) \ \frac{}{\text{tpm}(*\mathsf{T}, [], \emptyset)} \quad (\text{TPM-STAR}_2) \ \frac{\text{tpm}(\mathsf{T}, Q, \Gamma_1),\ \text{tpm}(*\mathsf{T}, LQ, \Gamma_2)}{\text{tpm}(*\mathsf{T}, Q \cdot LQ, \Gamma_1 \cup \Gamma_2)}$$

$$(\text{TPM-LIST}) \quad \frac{\text{tpm}(\mathsf{T}, Q, \Gamma_1),\ \text{tpm}(\mathsf{LT}, LQ, \Gamma_2)}{\text{tpm}(\mathsf{T} \cdot \mathsf{LT}, Q \cdot LQ, \Gamma_1 \cup \Gamma_2)}$$

$$(\text{TPM-UNION}) \quad \frac{\text{tpm}(\mathsf{T}_0, Q, \Gamma_0) \quad \text{or} \quad \text{tpm}(\mathsf{T}_1, Q, \Gamma_1)}{\text{tpm}(\mathsf{T}_0 + \mathsf{T}_1, Q, \Gamma)}, \quad \text{where:}$$

$$\Gamma = \begin{cases} \Gamma_0 + \Gamma_1 & \text{if } \text{tpm}(\mathsf{T}_0, Q, \Gamma_0) \text{ and } \text{tpm}(\mathsf{T}_1, Q, \Gamma_1) \\ \Gamma_i & \text{if } \text{tpm}(\mathsf{T}_i, Q, \Gamma_i) \text{ and for no } \Gamma' \ \text{tpm}(\mathsf{T}_{i+1 \bmod 2}, Q, \Gamma'),\ i = 0,1 \end{cases}$$

Table 6. Matching types and patterns.

$$(\text{TM-EMPTY}) \ \frac{}{\Gamma \vdash [] : []} \quad (\text{TM-TOP}) \ \frac{}{\Gamma \vdash M : \mathbf{T}}$$

$$(\text{TM-VALUE}) \ \frac{v : \mathsf{bt}}{\Gamma \vdash v : \mathsf{bt}} \quad (\text{TM-VAR}) \ \frac{\Gamma(x) < \mathsf{T}}{\Gamma \vdash x : \mathsf{T}}$$

$$(\text{TM-TAG}) \ \frac{\Gamma \vdash M : \mathsf{T}}{\Gamma \vdash f(M) : f(\mathsf{T})} \quad (\text{TM-LIST}) \ \frac{\Gamma \vdash M : \mathsf{T}, \quad \Gamma \vdash LM : \mathsf{LT}}{\Gamma \vdash (M \cdot LM) \ : \ (\mathsf{T} \cdot \mathsf{LT})}$$

$$(\text{TM-STAR}_1) \ \frac{}{\Gamma \vdash [] : *\mathsf{T}} \quad (\text{TM-STAR}_2) \ \frac{\Gamma \vdash M : \mathsf{T}, \quad \Gamma \vdash LM : *\mathsf{T}}{\Gamma \vdash (M \cdot LM) \ : \ *\mathsf{T}}$$

$$(\text{TM-UNION}) \quad \frac{\Gamma \vdash M : \mathsf{T} \quad \text{or} \quad \Gamma \vdash M : \mathsf{T}'}{\Gamma \vdash M : \mathsf{T} + \mathsf{T}'}$$

$$(\text{TM-ABS}) \quad \frac{\text{tpm}(\mathsf{T}, Q, \Gamma_1), \quad (\Gamma_1)_{|\tilde{x}} < \Gamma_Q, \quad (\Gamma_1)_{|\tilde{y}} > \Gamma_{|\tilde{y}}, \quad \Gamma, \Gamma_Q \vdash P : ok}{\Gamma \vdash (Q : \Gamma_Q)P : (\mathsf{T})\mathsf{Abs}}$$

where $\tilde{x} = \text{dom}(\Gamma_Q)$, $\tilde{y} = \text{fv}(Q) \setminus \tilde{x}$ and $(\Gamma_1)_{|\tilde{y}}$ is abstraction-free

Table 7. Type system for messages.

such abbreviations as $a.(f[?x:\mathsf{T}, ?y:\mathsf{T}'])P$ instead of $a.(f[x,y] : \{x:\mathsf{T}, y:\mathsf{T}'\})P$, and assume don't care variables '_' are always annotated with $\mathbf{T}$. Reduction semantics carries over to annotated closed processes formally unchanged.

In what follows, we shall use the following additional notation and terminology. We say that a type T is *abstraction-free* if T contains no subterms of the form $(\mathsf{T}')\mathsf{Abs}$. A context Γ is abstraction-free if for each $x \in \text{dom}(\Gamma)$, $\Gamma(x)$ is abstraction-free. We use $\Gamma \vdash u \in ch(\mathsf{T})$ as an abbreviation for: either $u = a \in ch(\mathsf{T})$ or $u = x \in \mathcal{V}$ and $\Gamma(x) = ch(\mathsf{T})$.

The type checking system, defined on open terms, consists of two sets of inference rules, one for messages and one for processes, displayed in Table 7 and 8, respectively. These two systems are mutually dependent, since abstractions may contain processes, and processes may contain abstractions. Note that the system is entirely syntax driven, i.e. the process P (resp. the pair (M, T)) determines the rule that should be applied to check $\Gamma \vdash P$ (resp. $\Gamma \vdash M : \mathsf{T}$).

The most interesting of these rules is (TM-ABS). Informally, $\Gamma \vdash A : (\mathsf{T})\mathsf{Abs}$ ensures that under Γ the following is true: (1) abstraction $A = (Q_{\tilde{x}} : \Gamma_Q)P$ behaves safely upon consuming messages of type T (because the type at which the actual parameters will be received is a subtype of the type declared for formal parameters, $(\Gamma_1)_{|\tilde{x}} < \Gamma_Q$, and because of $\Gamma, \Gamma_Q \vdash P : ok$);

$$(\text{T-In})\quad \frac{a \in ch(\mathsf{T}),\quad \Gamma \vdash A : (\mathsf{T})\mathsf{Abs}}{\Gamma \vdash a.A : ok}$$

$$(\text{T-Out})\ \frac{\Gamma \vdash u \in ch(\mathsf{T}),\quad \Gamma \vdash M : \mathsf{T}}{\Gamma \vdash \overline{u}\langle M\rangle : ok} \qquad (\text{T-Sum})\ \frac{\forall i \in I,\quad \Gamma \vdash a_i.A_i : ok \quad |I| \neq 1}{\Gamma \vdash \sum_{i\in I} a_i.A_i : ok}$$

$$(\text{T-Rep})\ \frac{\Gamma \vdash P : ok}{\Gamma \vdash\ !P : ok} \qquad (\text{T-Par})\ \frac{\Gamma \vdash P : ok,\quad \Gamma \vdash R : ok}{\Gamma \vdash\ (P|R) : ok}$$

$$(\text{T-Res})\ \frac{\Gamma \vdash P : ok}{\Gamma \vdash\ (\nu a)P : ok} \qquad (\text{T-Else})\ \frac{\Gamma \vdash P : ok,\quad \Gamma \vdash R : ok}{\Gamma \vdash\ P \text{ else } R : ok}$$

Table 8. Type system for processes.

(2) the pattern Q is consistent with type T, i.e. essentially the run-time type of Q is a subtype of T (because of type-pattern match and of $\Gamma_{|\tilde{y}} < (\Gamma_1)_{|\tilde{y}}$). This guarantees existence of a message of type T that matches the pattern. Moreover, no ill-formed pattern will arise from Q (abstraction-freeness).

Rule (T-In) checks that an abstraction A residing at channel $a \in ch(\mathsf{T})$ can safely consume messages of type T, and that there do exist messages of type T that match the pattern of A. Conversely (T-Out) checks that messages sent at u be of type T. Input and summation (rule (T-Sum)) are dealt with separately only for notational convenience. Finally, it is worth to notice that, by definition of $a : \mathcal{S}$, rule (TM-Value) entails subsumption on channels (i.e. $\Gamma \vdash a : \mathcal{S}$ and $\mathcal{S} < \mathcal{S}'$ implies $\Gamma \vdash a : \mathcal{S}'$). The remaining rules should be self-explanatory.

In the sequel, for closed annotated processes P, we shall write $P : ok$ for $\emptyset \vdash P : ok$, and say that P is well-typed. Similarly for $M : \mathsf{T}$, for closed annotated M.

Example 4. Assume $a \in ch(*\mathsf{int})$ and $b \in ch(f[\mathsf{int}, *\mathsf{int}])$. Then $P : ok$, where:

$$P = a.(?y : *\mathsf{int})b.(f[?x : \mathsf{int}, y])\overline{a}\langle x \cdot y\rangle \mid \overline{a}\langle[4,5]\rangle \mid \overline{a}\langle[4,5,6]\rangle.$$

Note that, if we change the sort of b into $ch(f[\mathsf{int},[\mathsf{int},\mathsf{int}]])$, then P is not well-typed, as rule (TM-Abs) fails on $A = (f[?x : \mathsf{int}, y])\overline{a}\langle x \cdot y\rangle$. This is intuitively correct, because a possible run-time type of A is $(f[\mathsf{int},[\mathsf{int},\mathsf{int},\mathsf{int}]])\mathsf{Abs}$, which is not consistent with the capacity associated to b, that is $f[\mathsf{int},[\mathsf{int},\mathsf{int}]]$.

To illustrate the use of $ch(\mathbf{T})$ and $ch(\mathbf{\bot})$, and contravariance on sort names, consider a "link process" ([7]) that constantly receives any *name* on a and sends it along b. This can be written as $!a.(?x : ch(\mathbf{\bot}))\overline{b}\langle x\rangle$. This process is well-typed provided $a \in ch(ch(\mathsf{T}))$, for some T, and that $b \in ch(ch(\mathbf{\bot}))$.

Remark 1 (on abstractions and subtyping). To see why we disallow subtyping on abstractions, consider the types $\mathsf{T} = [f(\mathsf{int}), f(\mathsf{int})]$ and $*f(\mathsf{int}) = \mathsf{T}'$. Clearly $\mathsf{T} < \mathsf{T}'$. Assume we had defined subtyping *covariant* on abstractions, so that $(\mathsf{T})\mathsf{Abs} < (\mathsf{T}')\mathsf{Abs}$. Now, clearly $A = (?x : \mathsf{T})\mathbf{0} : (\mathsf{T})\mathsf{Abs}$, but *not* $A : (\mathsf{T}')\mathsf{Abs}$ (the condition $(\Gamma_1)_{|\tilde{x}} < \Gamma_Q$ of (TM-Abs) fails). In other words, a crucial subtyping property would be violated.

On the other hand, assume we had defined subtyping *contravariant* on abstractions, so that $(\mathsf{T}')\mathsf{Abs} < (\mathsf{T})\mathsf{Abs}$. Consider $A' = (Q : \Gamma_Q)\mathbf{0}$, where $Q : \Gamma_Q = [f(?x : \mathsf{int}), f(?y : \mathsf{int}), f(?z : \mathsf{int})]$; clearly $A' : (\mathsf{T}')\mathsf{Abs}$, but *not* $A' : (\mathsf{T})\mathsf{Abs}$ (simply because there is no type-pattern match between T and Q). This would violate again the subtyping property.

Typing rules for Application and Case The rules below can be easily derived from the translation of derived constructs application and case to the base syntax. In the following, we let $\mathsf{T}_{M,\Gamma}$ denote the *exact type* of M under Γ, obtained from M by replacing each x by $\Gamma(x)$, each name $a \in ch(\mathsf{T})$ by $ch(\mathsf{T})$, each other v by the least type bt s.t. $v : \mathsf{bt}$, and, recursively, each abstraction subterm $(Q : \Gamma_Q)P$ by $(\mathsf{T}_{Q,\,\Gamma\cup\Gamma_Q})\mathsf{Abs}$. The rule for application is:

$$(\text{T-APPL}) \quad \frac{\Gamma \vdash A : (\mathsf{T}_{M,\Gamma})\mathsf{Abs}}{\Gamma \vdash A \bullet M : ok}.$$

that is easily proven sound recalling that $A \bullet M = (\nu c)(c.A|\overline{c}\langle M\rangle)$ (c fresh), and assuming that c is chosen s.t. $c \in ch(\mathsf{T}_{M,\Gamma})$.

Concerning Case, first note that the typed version of this construct contemplates annotated patterns, thus: Case M of $Q_1 : \Gamma_{Q_1} \Rightarrow P_1, \ldots, Q_k : \Gamma_{Q_k} \Rightarrow P_k : ok$. Then, relying on the rule for application, the typing rule for case can be written as:

$$(\text{T-CASE}) \quad \frac{\forall i = 1, \ldots, k : \ \Gamma \vdash (Q_i : \Gamma_{Q_i})P_i \bullet M : ok}{\Gamma \vdash \text{Case } M \text{ of } Q_1 : \Gamma_{Q_1} \Rightarrow P_1, \ldots, Q_k : \Gamma_{Q_k} \Rightarrow P_k : ok}.$$

Example 5 (a web service, continued). Consider the service defined in Example 2. Assume a basic type $\mathsf{mp3}$ of all mp3 files, such that $\mathtt{v}_{low}$, $\mathtt{v}_{high} : \mathsf{mp3}$, and a basic type $\mathsf{l\text{-}mp3}$ of low quality mp3 files, s.t. $\mathtt{v}_{low} : \mathsf{l\text{-}mp3}$, but *not* $\mathtt{v}_{high} : \mathsf{l\text{-}mp3}$. Assume $\mathsf{l\text{-}mp3} < \mathsf{mp3}$; note that this implies that $ch(\mathsf{mp3}) < ch(\mathsf{l\text{-}mp3})$, i.e. if a channel can be used for streaming generic files, it can also be used for streaming low-quality files, which fits intuition. Let T be `req_stream[bandwidth(string),channel(`ch`(mp3))]` and fix the following capacities for channels *stream* and *download*: $stream \in ch(\mathsf{T})$ and $download \in ch(\texttt{req\_down(}ch\texttt{((}\mathsf{T}\texttt{)Abs)))}$. An annotated version of WS, which permits in principle a static optimization of channels (assuming allocation of low-quality channels is less expensive than generic channels'):

$$\begin{aligned} WS = !(\quad & stream.(\texttt{req\_stream[bandwidth("low"),channel(?}x : ch(\mathsf{l\text{-}mp3})\texttt{)]})\overline{x}\langle \mathtt{v}_{low}\rangle \\ & + stream.(\texttt{req\_stream[bandwidth("high"),channel(?}y : ch(\mathsf{mp3})\texttt{)]})\overline{y}\langle \mathtt{v}_{high}\rangle \\ & + download.(\texttt{req\_down[?}z : ch(\texttt{((}\mathsf{T}\texttt{)Abs)]})\overline{z}\langle Player\rangle \quad) \end{aligned}$$

where *Player* is the obvious annotated version of the player of Example 2. It is easy to check that $Player : (\mathsf{T})\mathsf{Abs}$ and that $WS : ok$.

4 Run-Time Safety

The safety property of our interest can be defined in terms of channel capacities, message types, and consistency. First, a formal definition of pattern consistency.

Definition 8 (T-consistency). *A type T is* consistent *if $\mathbb{L}$ does not occur in T. A pattern Q is* T-consistent *if there is a message $M : T$ that matches Q.*

Note that all sort names, including $ch(\mathbb{L})$, are consistent types by definition. A safe process is one whose output and input actions are in agreement with channel capacities, as stated by the definition below. Of course, for input actions it makes sense to require consistency (condition 2) only if the input channel has in turn a consistent capacity.

Definition 9 (safety). *Let P be an annotated closed process. P is* safe *if and only if for each name $a \in ch(T)$:*

1. *whenever $P \equiv (\nu \tilde{h})(\overline{a}\langle M\rangle \,|\, R)$ then $M : T$;*
2. *suppose T is consistent. Whenever $P \equiv (\nu \tilde{h})(S \,|\, R)$, where S is a guarded summation, $a.A$ a summand of S and Q is A's pattern, then Q is T-consistent.*

Theorem 1 (run-time safety). *Let P be a closed annotated process. If $P : ok$ and $P \rightarrow^* P'$ then P' is safe.*

5 Dynamic abstractions

Although satisfactory in most situations, a static typing scenario does not seem appropriate in those cases where little is known in advance on actual types of data that will be received from the network.

Example 6 (a directory of services). Suppose one has to program an online directory of (references to) services. Upon request of a service of type T, for *any* T, the directory should lookup its catalog and respond by sending a channel of type $ch(\mathsf{T})$ along a reply channel. If the reply channel is fixed statically, it must be given capacity $ch(\mathbb{1})$, that is, any channel. Then, a client that receives a name at this channel must have some mechanism to cast at runtime this generic type to the subtype $ch(\mathsf{T})$, which means going beyond static typing. If the reply channel is provided by clients the situation does not get any better. E.g. consider the following service (here we use some syntactic sugar for the sake of readability):

$$!\mathit{request}.(\mathtt{req[?}t:\mathsf{Td}\mathtt{,?}x_{rep}:ch(\mathsf{Tr})\mathtt{]}\,)\,\mathsf{let}\,y=\mathit{lookup}(t)\,\mathsf{in}\,\overline{x_{rep}}\langle y\rangle$$

where lookup is a function from some type Td of type-descriptors to the type of *all* channels, $ch(\mathbb{1})$. It is not clear what capacity Tr the return channel variable x_{rep} should be assigned. The only choice that makes the above process well typed is to set $\mathsf{Tr}=ch(\mathbb{1})$, that is, x_{rep} can transport any channel. But then, a client's call to this service like $\overline{\mathit{request}}\langle\mathtt{req[}v_{td}\mathtt{,}r\mathtt{]}\rangle$, where r has capacity $ch(\mathsf{T})$, is not well typed (because $r\in ch(ch(\mathsf{T}))$ and $ch(ch(\mathsf{T}))$ is not a subtype of $ch(\mathsf{Tr})=ch(ch(\mathbb{1}))$).

Even ignoring the static vs. dynamic issue, the schemas sketched above would imply some form encoding of type and subtyping into XML, which is undesirable if one wishes to reason at an abstract level. As we shall see below, dynamic abstractions can solve these difficulties.

The scenario illustrated in the above example motivates the extension of the calculus presented in the preceding sections with a form of dynamic abstraction. The main difference from ordinary abstractions is that type checking for pattern variables is moved to run-time. This is reflected into an additional communication rule, that explicitly invokes type checking. We describe below the necessary extensions to syntax and semantics. We extend the syntactic category of Abstractions thus:

$$A ::= \cdots \mid (\!|Q_{\tilde{x}}:\Gamma|\!)P \;\; \textit{Dynamic abstraction}$$

with $\tilde{x}=\mathsf{dom}(\Gamma)$. We let D range over dynamic abstractions and A over all abstractions. We add a new reduction rule:

$$(\textsc{com-d})\quad \frac{j\in I,\;\; a_j=a,\quad A_j=(\!|Q_{\tilde{x}}:\Gamma|\!)P,\quad \mathsf{match}(M,Q,\sigma),\quad \forall y\in\mathsf{dom}(\sigma):\;\sigma(y):\Gamma(y)}{\overline{a}\langle M\rangle\mid\sum_{i\in I}a_i.A_i\to P\sigma}.$$

We finally add a new type checking rule. For this, we need the following additional notation. Given Γ_1 and Γ_2, we write $\Gamma_1\lesseqgtr\Gamma_2$ if $\mathsf{dom}(\Gamma_1)=\mathsf{dom}(\Gamma_2)$ and $\forall x\in\mathsf{dom}(\Gamma_1)$ there is a consistent type T s.t. $\mathsf{T}<\Gamma_1(x)$ and $\mathsf{T}<\Gamma_2(x)$.

$$(\textsc{tm-abs-d})\;\frac{\mathsf{tpm}(\mathsf{T},Q,\Gamma_1),\quad (\Gamma_1)_{|\tilde{x}}\lesseqgtr\Gamma_Q,\quad (\Gamma_1)_{|\tilde{y}}>\Gamma_{|\tilde{y}},\quad \Gamma,\Gamma_Q\vdash P:ok}{\Gamma\vdash(\!|Q_{\tilde{x}}:\Gamma_Q|\!)P:(\mathsf{T})\mathsf{Abs}}$$

where $\widetilde{y}=\mathsf{fv}(Q)\setminus\widetilde{x}$ and $(\Gamma_1)_{\lfloor\tilde{y}}$ is abstraction free. The existence of a common consistent subtype for Γ_Q and $(\Gamma_1)_{|\tilde{x}}$ ensures a form of dynamic consistency for Q, detailed below.

We discuss now the extension of run-time safety. The safety property needs to be extended to inputs formed with dynamic abstractions. A stronger form of pattern consistency is needed.

Definition 10 (dynamic T-consistency). *An annotated pattern* $Q : \Gamma$ $(\mathrm{fv}(Q) = \mathrm{dom}(\Gamma))$ *is* dynamically T-consistent *if there is a message* $M : T$ *s.t.* $\mathrm{match}(Q, M, \sigma)$ *and* $\forall x \in \mathrm{dom}(\sigma)$ *we have* $\sigma(x) : \Gamma(x)$.

Definition 11 (dynamic safety). *Let* P *an annotated closed process.* P *is* dynamically safe *if for each name* $a \in ch(T)$ *conditions 1 and 2 of Definition 9 hold, and moreover the following condition is true: Suppose* T *is consistent. Whenever* $P \equiv (\nu \tilde{h})(S \mid R)$*, where* S *is a guarded summation,* $a.D$ *is a summand of* S *and* $Q : \Gamma$ *is* D*'s annotated pattern, then* $Q : \Gamma$ *is dynamically* T*-consistent.*

Theorem 2 (run-time dynamic safety). *Let* P *be an annotated closed process in the extended language. If* $P : ok$ *and* $P \rightarrow^* P'$ *then* P' *is dynamically safe.*

Example 7 (a directory of services, continued). Consider again the directory of services. Clients can either request a (reference to a) service of a given type, by sending a message to channel *discovery*, or request the directory to update its catalog with a new service, using the channel *publish*. Each request to *discovery* should contain some type information, which would allow the directory to select a (reference to a) service of that type, taking subtyping into account. Types cannot be passed around explicitly. However one can pass a dynamic abstraction that will do the selection on behalf of the client and return the result back to the client at a private channel. The catalog is maintained on a channel *cat* local to the directory. Thus the directory process can be defined as follows, where $\prod_{i \in I} ! \overline{cat}\langle c_i \rangle$ stands for $! \overline{cat}\langle c_1 \rangle \mid \cdots \mid ! \overline{cat}\langle c_n \rangle$ (for $I = 1, \ldots, n$) and the following capacities are assumed: $discovery \in ch((ch(\mathbf{1}))\mathsf{Abs}, publish, cat \in ch(\mathsf{ch}(\mathbf{1}))$.

$$Directory \stackrel{\triangle}{=} (\nu\, cat)(\textstyle\prod_{i \in I} ! \overline{cat}\langle c_i \rangle \mid ! publish.(?y : ch(\mathbf{1}))! \overline{cat}\langle y \rangle \\ \mid ! discovery.(?x : (ch(\mathbf{1}))\mathsf{Abs})\, cat.x\,)$$

Note that $(ch(\mathbf{1}))\mathsf{Abs}$ is the type of all abstractions that can consume some channel. A client that wants to publish a new service S that accepts messages of some type T at a new channel $a \in ch(\mathsf{T})$ is:

$$C_1 \stackrel{\triangle}{=} (\nu\, a)(\overline{publish}\langle a \rangle \mid S\,).$$

A client that wants to retrieve a reference to a service of type T, or any subtype of it, is:

$$C_2 \stackrel{\triangle}{=} (\nu\, r)(\overline{discovery}\langle (\!| ?z : ch(\mathsf{T}) |\!) \bar{r}\langle z \rangle \rangle \mid r.(?y : ch(\mathsf{T}))C'\,).$$

Suppose $r \in ch(ch(\mathsf{T}))$. Assuming S and C' are well typed (the latter under $\{y : ch(\mathsf{T})\}$), it is easily checked that the global system

$$P \stackrel{\triangle}{=} Directory \mid C_1 \mid C_2$$

is well typed too.

In reality, the above solution would run into security problems, as the directory executes blindly any abstraction received from clients ($cat.x$). Moreover, services originating from unauthorized clients should not be published. We can avoid these problems using encryption so to authenticate both abstractions and published services. We rely on the encoding of encryption primitives[2] described in Section 2. Assume that every client C_j shares a secret key k_j with the directory. A table associating clients identifiers and keys is maintained on a channel *table* local to the directory (hence secure). Assume that identifiers $id_j, \ldots$ are of a basic type identifier, that keys $k_j, \ldots$ are names of a sort Key and let $\mathsf{enc}(\mathsf{T})$ be the type of messages $\{M\}_k$ where $M : \mathsf{T}$. Fix the following capacities: $cat \in ch(ch(\mathbf{1}))$, $table \in ch(\texttt{[id(identifier),key(Key)]})$, $publish \in ch(\texttt{service\_p[id(identifier),channel(enc}(ch(\mathbf{1}))\texttt{)]})$, and $discovery \in ch(\texttt{service\_d[id(identifier),abstr(enc}((ch(\mathbf{1}))\mathsf{Abs})\texttt{)]})$. The process $Directory_s$ is:

[2] For the purpose of the present example, we extend the encoding to the typed calculus by $[\![\{M\}_k]\!] \stackrel{\triangle}{=} ([k, ?x : ch(\mathbf{T})])\bar{x}\langle [\![M]\!] \rangle$, and $\langle\!|\, \mathrm{case}\, M \,\mathrm{of}\, \{x : \mathsf{T}\}_k \,\mathrm{in}\, P \,|\!\rangle \stackrel{\triangle}{=} (\nu\, r)\,([\![M]\!] \bullet [k, r] \mid r.(\!| ?x : \mathsf{T} |\!)\, \langle\!| P |\!\rangle)$, with $r \in ch(\mathbf{T})$.

$$\begin{aligned}\mathit{Directory}_s \triangleq (\nu\, cat, table)\Big(&\textstyle\prod_{i\in I}!\overline{cat}\langle c_i\rangle \mid \prod_{j\in J}!\overline{table}\langle \texttt{[id(}id_j\texttt{),key(}k_j\texttt{)]}\rangle\\ &\mid !\,publish.(\texttt{service\_p[id(?}x:\mathsf{identifier}\texttt{),channel(?}z_c:\mathsf{enc}(ch(\mathbf{1}))\texttt{)]})\\ &\quad table.(\texttt{[id(}x\texttt{),key(?}x_k:\mathsf{Key}\texttt{)]})\,\mathrm{case}\,z_c\,\mathrm{of}\,\{y:ch(\mathbf{1})\}_{x_k}\,\mathrm{in}\,!\overline{cat}\langle y\rangle\\ &\mid !\,discovery.(\texttt{service\_d[id(?}x:\mathsf{identifier}\texttt{),abstr(?}z_a:\mathsf{enc}((\mathsf{ch}(\mathbf{1}))\mathsf{Abs})\texttt{)]})\\ &\quad table.(\texttt{[id(}x\texttt{),key(?}x_k:\mathsf{Key}\texttt{)]})\,\mathrm{case}\,z_a\,\mathrm{of}\,\{y:(\mathsf{ch}(\mathbf{1}))\mathsf{Abs}\}_{x_k}\,\mathrm{in}\,cat.y\Big)\end{aligned}$$

The client C_1 may be rewritten as:

$$C_1' \triangleq (\nu\, a)(\,\overline{publish}\langle\texttt{service\_p[id(}id_1\texttt{),channel(}\{a\}_{k_1}\texttt{)]}\rangle \mid S\,)$$

and C_2 as:

$$C_2' \triangleq (\nu\, r)(\,\overline{discovery}\langle\texttt{service\_d[id(}id_2\texttt{),abstr(}\{(\!|?z:ch(\mathsf{T})|\!)\bar{r}\langle z\rangle\}_{k_2}\texttt{)]}\rangle \mid r.(?y:ch(\mathsf{T}))C'\,).$$

Suppose $a \in ch(\mathsf{T}')$, $r \in ch(ch(\mathsf{T}))$ and assume S and C' are well typed under the appropriate contexts. The global system

$$P_s \triangleq (\nu k_1, k_2)(\mathit{Directory}_s \mid C_1' \mid C_2')$$

is well typed too. An attacker may intercept messages on *publish* or *discovery* and may learn the identifiers of the clients, but not the secret shared keys. As a consequence, it cannot have $\mathit{Directory}_s$ publish unauthorized services or run unauthorized abstractions.

6 Conclusions and related work

XPi's type system can be extended into several directions. We are presently considering types that would guarantee "responsiveness" of services. A responsive service would be one that, when invoked at a given a, eventually responds at a given return address r, possibly after collaborating with other services that are equally responsive. This extension would be along the lines of Sangiorgi's *uniform receptiveness* [21]. Such a system might be augmented with primitives for managing quality of service in terms of response time.

A number of proposals aim at integrating XML processing primitives in the context of traditional, statically typed languages and logics. The most related to our work are XDuce [16] and $\mathbb{C}$Duce, [3], two typed (functional) languages for XML document processing. XPi's list-like representation of documents draws its inspiration from them. TQL [9] is both a logic and a query language for XML, based on a spatial logic for the Ambient calculus [10]. All these languages support query primitives more sophisticated than XPi's patterns, but issues raised by communication and code/name mobility, which are our main focus, are of course absent.

Early works aiming at integration of XML into process calculi, or vice-versa, are [14] and [4]. Xdπ [14] is a calculus for describing interaction between data and processes across distributed locations; it is focused on process migration rather than communication. A type system is not provided. Iota [4] is a concurrent XML scripting language for home-area networking. It relies on syntactic subtyping, like XPi, but is characterized by a different approach to XML typing. In particular, Iota's type system just ensures well-formedness of XML documents, rather than the stronger validity, which we consider here.

Roughly contemporary to ours, and with similar goals, are [8] and [11]. The language πDuce of [8] features asynchronous communication and code/name mobility. Similarly to XDuce's, πDuce's pattern matching embodies built-in type checks, which may be expensive at run-time. The language in [11] is basically a π-calculus enriched with a rich form of "semantic" subtyping and pattern matching. Code mobility is not addressed. Pattern matching, similarly to πDuce's, performs type checks on messages. By contrast, in XPi static type checks and plain pattern

matching suffice, as types of pattern variables are checked statically against channel capacities. We confine dynamic type checking to dynamic abstractions, which can be used whenever no refined typing information on incoming messages is available (e.g. at channels of capacity **T**). Both [11] and [8] type systems also guarantee a form of absence of deadlock, which however presupposes that basic values do not appear in patterns. In XPi, we thought it was important to allow basic values in patterns for expressiveness reasons (e.g., they are crucial in the encoding of the spi-calculus presented in Section 2).

References

1. M. Abadi and A.D. Gordon. A Calculus for Cryptographic Protocols: The Spi Calculus. *Information and Computation*, 148(1):1-70, Academic Press, 1999.
2. T. Andrews, F. Curbera, and S. Thatte. Business Process Execution Language for Web Wervices, v1.1, 2003. `http://ifr.sap.com/bpel4ws`.
3. V. Benzaken, G. Castagna, and A. Frisch. Cduce: An XML-Centric General-Purpose Language. In *Proceedings of the ACM International Conference on Functional Programming*, 2003.
4. G.M. Bierman and P. Sewell. Iota: A concurrent XML scripting language with applications to Home Area Networking. Technical Report 577, University of Cambridge Computer Laboratory, 2003.
5. Biztalk Server Home. `http://www.microsoft.com/biztalk/`.
6. S. Bjorg and L.G. Meredith. Contracts and Types. *Communication of the ACM*, 46(10), October 2003.
7. M. Boreale. On the Expressiveness of Internal Mobility in Name-Passing Calculi. *Theoretical Computer Science*, 195, 1998.
8. A. Brown, C. Laneve, and L.G. Meredith. πDuce: A process calculus with native XML datatypes. Manuscript. 2004.
9. L. Cardelli and G. Ghelli. TQL: A Query Language Semistruictured Data Based on the Ambient Logic. *Mathematical Structures in Computer Science*, 14:285–327, 2004.
10. L. Cardelli and A.D. Gordon. Mobile ambients. *Theoretical Computer Science*, 240(1), 2000.
11. G. Castagna, R. De Nicola, and D. Varacca. Semantic subtyping for the π-calculus. To appaear in *Proc. of LICS'05*.
12. E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language 1.1. W3C Note, 2001. `http://www.w3.org/TR/2001/NOTE-wsdl-20010315`.
13. D.C. Fallside. XML Schema Part 0: Primer. W3C Recommendation, 2001. `http://www.w3.org/TR/2001/REC-xmlschema-0-20010502`.
14. P. Gardner and S. Maffeis. Modeling Dynamic Web Data. In *Proceedings of DBPL 2003*, volume 2921 of *LNCS*. Springer, 2003.
15. H. Hosoya and B. Pierce. Regular Expression Pattern Matching for XML. *Journal of Functional Programming*, 2002.
16. H. Hosoya and B. Pierce. Xduce: A Statically Typed XML Processing Language. In *Proceedings of ACM Transaction on Internet Technology*, 2003.
17. M. Merro. Locality and polyadicity in asynchronous name-passing calculi. In *Proceedings of FoSSaCS 2000*, volume 1784 of *LNCS*, pages 238–251. Springer, 2000.
18. R. Milner. The Polyadic π-Calculus: a Tutorial. Technical Report ECS-LFCS-91-180, LFCS, Dept. of Computer Science, Edinburgh University, 1991.
19. R. Milner, J. Parrow, and D. Walker. A calculus of Mobile Processes, part I and II. *Information and Computation*, 100:1–41 and 42–78, 1992.
20. B. Pierce and D. Sangiorgi. Typing and Subtyping for Mobile Process. *Mathematical Structures in Computer Science*, 6(5), 1996.
21. D. Sangiorgi. The name discipline of uniform receptiveness. *Theoretical Computer Science*, 221, 1999.
22. D. Sangiorgi and R. Milner. Barbed bisimulation. *Proc. of Concur'92*, LNCS, Springer, 1992.
23. Web services activity web site, 2002. `http://www.w3.org/2002/ws`.